Compiling Safe Languages To Native Code

David Tarditi
Principal Researcher
Advanced Compiler Technology Group

Safe languages

- Enforce safety at compile and run time
  - Type safe: values used only at their declared type
  - Memory safe: programs write only in their own (allocated) memory
  - No undefined behavior
  - Examples: C#, Java, Modula-3, Cedar/Mesa
- Benefits
  - Higher programmer productivity
  - Detect errors at compile or run time leading to common security attacks
    - Buffer overruns, stack smashing, double de-allocate
  - More reliable programs

Native code

- Machine code that runs directly on a processor
  - Example: x86 code
- Typically used for:
  - OS-level programming (Windows)
  - Commercial desktop applications (Office)
  - Web server/browser implementations (Internet)

The problem

- Safe languages not usable for native code development
  - Safe programs compiled to bytecode
  - Bytecode runs within a runtime system (CLR or JVM)
- How can we use safe languages to create native code programs?
  - C# -> x86 instead of C++ -> x86?

Bartok

- System we developed that shows how you can use safe languages for native code.
  - MSIL -> x86 / x64
  - Focus is on supporting systems programming
- Consists of:
  - Ahead-of-time compiler
  - Lightweight runtime library (32 KLOC)

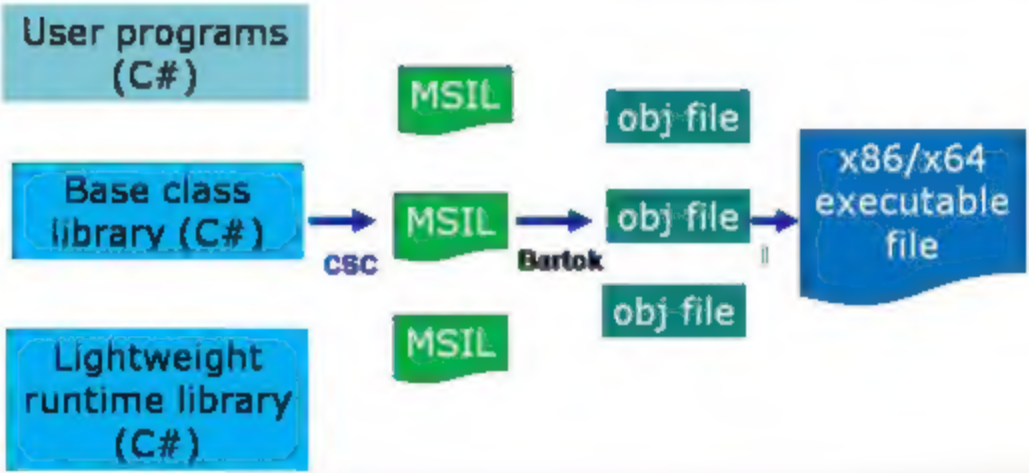
Supporting systems programming

- Focus is supporting systems programming
  - Bartok runtime library written mostly in C#
- Bartok is the compiler and runtime library for Singularity
- Singularity is a research OS written almost entirely in C#
  - See http://singularity for more information

Outline

- **Overview of Bartok**
- Performance results
- Conclusion

Conventional compiler model



Compiler

- Optimizing ahead-of-time compiler
- Supports a range of compilation modes
  - Per-function, per-assembly, whole program
- Optimizations include
  - Advanced interprocedural opts
  - "Textbook" per-procedure optimizations
  - Inlining
- Non-goal: product quality code generation

Interprocedural optimizations

- Optimizations that cross methods
  - Class to whole program in scope
- Affects patching story
  - Patching/opt granularity must be same
- Benefits:
  - Eliminate redundant safety checks
    - Null checks, type casts, bounds checks
  - Improve OO programs
    - Devirtualization, tree shaking

Interprocedural optimizations

- Bartok optimizations not usually found in production compilers
- C++:
  - Safety check optimizations not done because
    - Programs are unsafe (no bounds checks)
    - Programmers do optimizations by hand (static_cast)
  - OO optimizations developed for type-safe settings
    - Lack of language type safety complicates implementing them
- C#: too costly to do during just-in-time compilation

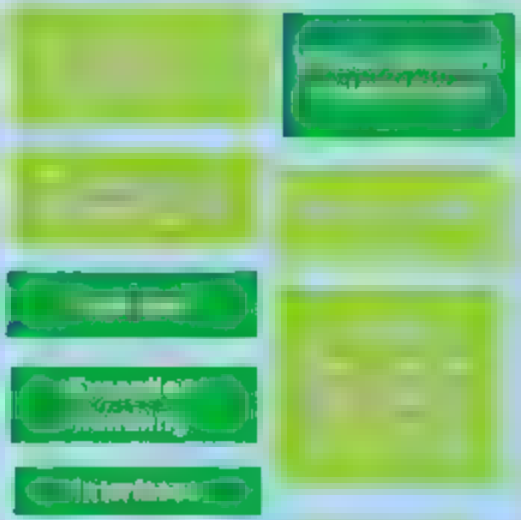
Modular, lightweight runtime library

Modular, lightweight runtime library

Code size:

- GCs: 20K lines
- Threading: 2K lines
- Type tests: 800 lines
- EH: 600 lines
- V-Tables, object layout: 2.5K
- P/Invoke: 500 lines
- Core datatypes: 5K lines

Total: ~32K lines



Current limitations

This is a research prototype, not a product-quality system.

- Not as rich an environment as CLR
  - Porting .NET libraries on demand
- DLLs not implemented
- No COM interop
- No CAS
- Minimal support for reflection

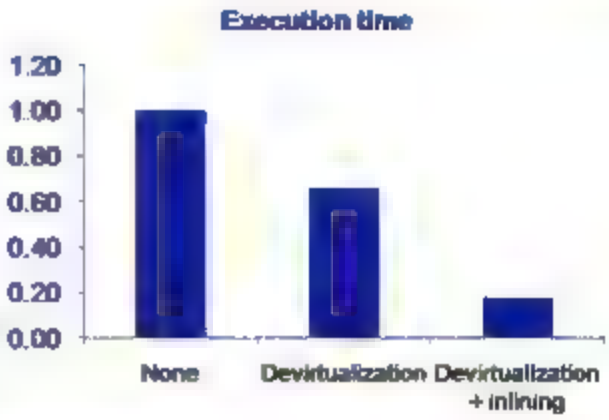
Outline

- Overview of Bartok
- **Performance results**
- Conclusion

Effect of devirtualization


```
class A {
    public virtual int F() {
        return 1;
    }
}

A obj = new A();
int sum=0;
for (int i=0; i<bound; i++) {
        sum += obj.F();
}
```



Tree shaking code size reductions for Singularity components

Program	Code Whole	Code w/ Tree Shake	% Reduction
Kernel	2.37 MB	1.29 MB	46%
IDE Disk Driver	1.85 MB	455 KB	75%
Web Server	2.73 MB	765 KB	72%
Content Extension	2.14 MB	502 KB	77%

Memory footprint

Memory footprint
"Hello World" process

	Singularity	Windows XP (SP2)	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)
C - static lib		544K	232K	664K
C++ - static lib		572K	704K	1,216K
C# - w/ GC	408K	3,750K (CLR)		

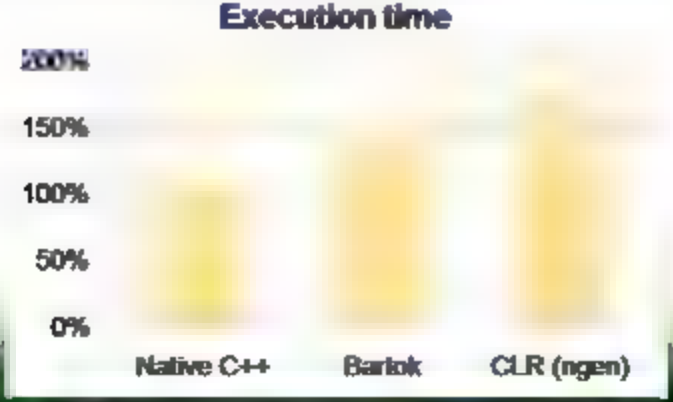
- C# process w/ GC can have similar memory footprint to C++ w/ heap

Real-world app case study

- Phoenix is Microsoft's new compiler and programming tools infrastructure
- Bartok compiled managed Phoenix backend
  - Phoenix compiles to 100% native code or 100% managed code
- Total of 1.5 million lines of code (including synthesized code)

Results

- Resulting backend can compile NT kernel, ~2600 regress suite files
- Performance of Phoenix backend, averaged over the ~2600 regress files



Analysis of results

- Analyzed where time goes compiling GCC (138% of native)
- Can be improved with product-quality GC, code generator

Technology transfer

- Target for compiler is Phoenix
  - Same optimization technology for C++/native and C#/native
  - Plus additional Bartok opts
- DevDiv and MSR working together to integrate Bartok compiler w/ Phoenix
  - Goal: support incubation efforts
  - Virtual team of 5 devs
- Targeting Bartok runtime library

Conclusion

- Bartok allows you to write native code in safe languages like C#:
  - Optimizing ahead-of-time compiler
  - Modular runtime library
- Brings benefits of safe languages to native code

Follow up

- For more information:
  - http://sharepoint/sites/bartok
  - http://singularity
- We plan to make a binary drop of Bartok available for internal experimentation
  - Email me if interested (alias: dtarditi)

Questions?